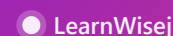


Aplicaciones en tiempo real con push del servidor — Libro de lecciones

Todas las lecciones de los módulos de este curso, reunidas en un solo documento.



Índice

01	Arquitectura web en tiempo real en Wisej.NET	Módulo 1 · Guía de la lección
02	Arquitectura web en tiempo real en Wisej.NET	Módulo 1 · Guía del laboratorio / examen
03	Contexto de la aplicación, sesiones y ciclo de vida	Módulo 2 · Guía de la lección
04	Contexto de la aplicación, sesiones y ciclo de vida	Módulo 2 · Guía del laboratorio / examen
05	Push del servidor para una sesión con tareas en segundo plano	Módulo 3 · Guía de la lección
06	Push del servidor para una sesión con tareas en segundo plano	Módulo 3 · Guía del laboratorio / examen
07	Temporizadores, sondeo de respaldo y cadencia de actualización	Módulo 4 · Guía de la lección
08	Temporizadores, sondeo de respaldo y cadencia de actualización	Módulo 4 · Guía del laboratorio / examen
09	Enlace de datos en vivo y cuadrículas en tiempo real	Módulo 5 · Guía de la lección
10	Enlace de datos en vivo y cuadrículas en tiempo real	Módulo 5 · Guía del laboratorio / examen
11	Push multiusuario con servicios y hubs de eventos	Módulo 6 · Guía de la lección
12	Push multiusuario con servicios y hubs de eventos	Módulo 6 · Guía del laboratorio / examen
13	Refuerzo para producción, despliegue y proyecto final	Módulo 7 · Guía de la lección
14	Refuerzo para producción, despliegue y proyecto final	Módulo 7 · Guía del laboratorio / examen

Arquitectura web en tiempo real en Wisej.NET

Arquitectura en tiempo real dirigida por el servidor: push por WebSocket, actualizaciones dentro de la petición y polling como alternativa en el modelo del lado del servidor de Wisej.NET. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Arquitectura en tiempo real dirigida por el servidor: push por WebSocket, actualizaciones dentro de la petición y polling como alternativa en el modelo del lado del servidor de Wisej.NET.

Por qué existe este módulo

Las aplicaciones web en tiempo real suelen asociarse con chats, juegos, cotizaciones bursátiles, pantallas de monitorización y dashboards. En los sistemas de gestión empresarial la misma necesidad aparece de formas menos espectaculares pero más valiosas: un supervisor ve los tickets nuevos a medida que llegan, un operador de producción ve que una máquina se detiene, un despachador ve que un trabajo se ha reasignado o un administrativo de facturación ve que una importación sigue en curso sin pulsar Refresh.

Wisej.NET se basa en un modelo de aplicación del lado del servidor. El desarrollador trabaja en C# o VB.NET y usa controles, eventos, formularios, páginas, temporizadores, enlace de datos y el diseñador de Visual Studio, mientras Wisej.NET mantiene el árbol de widgets del navegador y lo sincroniza con el estado del servidor. Este modelo es especialmente importante para el push del servidor, porque el servidor ya es el dueño del estado de la aplicación. Una tarea en segundo plano puede cambiar una etiqueta, añadir una fila a una cuadrícula, actualizar una barra de progreso o abrir una notificación, y Wisej.NET puede entregar los cambios de interfaz pendientes al navegador cuando existe una conexión WebSocket activa.

Un curso de nivel principiante suele enseñar a crear páginas, ventanas, controles, menús y manejadores de eventos básicos. Este curso de nivel intermedio da por sabidas esas habilidades y se centra en las cuestiones de diseño de producción que surgen cuando la interfaz se actualiza sola. La pregunta central no es «¿Puedo actualizar una etiqueta desde un hilo?». La pregunta central es «¿Cómo actualizo la sesión del usuario correcto, al ritmo adecuado, desde el servicio adecuado, sin perder estado, sin fugas de memoria, sin bloquear el servidor y sin saturar el navegador?».

Qué significa «tiempo real» en este curso

En este curso, una aplicación Wisej.NET en tiempo real es una aplicación que puede recibir eventos del usuario con poca latencia y que, además, puede actualizar el navegador del usuario desde el servidor sin esperar otro clic. El push del servidor es la parte importante. La aplicación debe poder cambiar la interfaz visible porque algo ha ocurrido en el servidor: un trabajo en segundo plano ha avanzado, una cola ha cambiado, ha llegado un mensaje u otro usuario ha realizado una acción.

No reduzcas el tiempo real a un «refresco rápido». Una cuadrícula que se refresca cada 30 segundos a veces es aceptable, pero no es lo mismo que el push. El push está dirigido por eventos: la aplicación se actualiza porque el modelo ha cambiado. El polling pregunta una y otra vez si algo ha cambiado. Ambos pueden ser útiles, pero tienen un coste, una latencia y un comportamiento ante fallos distintos.

El modelo mental del tiempo real en Wisej.NET

Un modelo mental útil es este:

1. El navegador aloja los widgets JavaScript de Wisej.NET.
2. El servidor aloja la sesión de la aplicación Wisej.NET y los controles del lado del servidor.
3. Las acciones del usuario viajan del navegador al servidor como eventos.
4. El código del servidor modifica los controles y el estado de la aplicación.
5. Wisej.NET envía los cambios de vuelta al navegador como actualizaciones compactas.
6. Con WebSocket, una tarea en segundo plano puede enviar las actualizaciones pendientes sin esperar la siguiente petición del navegador.

Esto no es screen scraping ni un escritorio remoto. El navegador sigue renderizando widgets web nativos y el servidor no transmite píxeles. El punto arquitectónico importante es que la lógica de la aplicación y el estado autoritativo de la interfaz viven en el servidor. Cuando cambia una propiedad de un control de Wisej.NET, el widget del navegador puede sincronizarse.

Push por WebSocket frente a actualizaciones dirigidas por peticiones

Una petición web normal empieza cuando el navegador contacta con el servidor y termina cuando el servidor responde. Si no hay un canal abierto del servidor al navegador, el servidor no puede contactar con el navegador por iniciativa propia: solo puede enviar cambios cuando el navegador hace otra petición. WebSocket cambia eso. Con una conexión WebSocket, Wisej.NET puede enviar actualizaciones de interfaz al cliente conectado desde el código del servidor.

En Wisej.NET, una tarea en segundo plano puede actualizar controles de la interfaz y llamar a `Application.Update()` para enviar los cambios pendientes cuando WebSocket está disponible. Si WebSocket no está disponible, `Application.Update()` no puede hacer push porque no hay una conexión activa. La aplicación necesita entonces una alternativa, como el polling de Wisej.NET o peticiones dirigidas por temporizador. La alternativa no invalida la aplicación: simplemente cambia el mecanismo de entrega.

Tres mecanismos de actualización que debes distinguir

1. Actualizaciones dentro de la petición

Es el caso más sencillo. El usuario hace clic en un botón, se ejecuta un manejador de eventos, el manejador cambia los controles y Wisej.NET devuelve los cambios con la respuesta. Aquí normalmente no llamas a `Application.Update()`, porque la actualización vuelve al navegador de forma natural al final de la petición.

```
private void buttonRefresh_Click(object sender, EventArgs e)
{
    statusLabel.Text = "Refreshed at " + DateTime.Now.ToString();
}
```

2. Push por WebSocket fuera de la petición

Este es el tema principal del curso. Una tarea en segundo plano se ejecuta fuera del evento original del navegador. Modifica la interfaz del lado del servidor y llama a `Application.Update()` para enviar los cambios a través de la conexión WebSocket.

```
private void buttonStart_Click(object sender, EventArgs e)
{
    buttonStart.Enabled = false;
    progressBar.Value = 0;

    Application.StartTask(() =>
    {
        for (int i = 1; i ≤ 100; i++)
        {
            progressBar.Value = i;
            statusLabel.Text = $"Processing {i}%";
            Application.Update(this);
            Thread.Sleep(100);
        }

        buttonStart.Enabled = true;
        statusLabel.Text = "Done";
        Application.Update(this);
    });
}
```

3. Polling o refresco dirigido por temporizador

Cuando un navegador o un servidor no pueden usar WebSocket, el servidor no puede enviar cambios fuera de la petición. En ese caso, la aplicación puede usar polling para que el navegador contacte periódicamente con el servidor y reciba los cambios pendientes. También se puede usar un `Wisej.Web.Timer` para disparar eventos del lado del servidor a un intervalo definido.

```
private void MainPage_Load(object sender, EventArgs e)
{
    // Ignored when WebSocket is available. Useful as fallback when it is not.
    Application.StartPolling(1000);
}
```

El curso enseña a diseñar primero para WebSocket y después degradar de forma predecible.

El proyecto del curso: TicketOps Live

Todos los módulos construyen partes de la misma aplicación, **TicketOps Live**. Es una consola de operaciones de soporte en tiempo real con estas funciones:

- una franja de estado del sistema en directo;
- un monitor del progreso de los trabajos del servidor;
- un tablero de tickets que se actualiza cuando se crean o modifican tickets;

- un panel de notificaciones para eventos multiusuario;
- un dashboard de operaciones con indicadores sencillos de salud del servidor;
- una lista de comprobación de producción para el despliegue detrás de IIS o de un balanceador de carga.

El proyecto es deliberadamente sencillo desde el punto de vista del negocio. El objetivo es aprender arquitectura en tiempo real, no implementar un sistema de helpdesk completo. Cada laboratorio debe dejar al estudiante un código reutilizable en aplicaciones reales.

Regla de diseño: primero el modelo, después la interfaz

Es tentador dejar que las tareas en segundo plano manipulen directamente cada control que debe cambiar. Funciona en demos pequeñas, pero las aplicaciones reales necesitan separar los datos de la presentación. Normalmente la tarea en segundo plano debe actualizar un modelo o lanzar un evento de dominio, y la interfaz específica de la sesión debe decidir cómo mostrarlo.

En una ventana de progreso de un solo usuario, actualizar los controles directamente es aceptable. En los dashboards multiusuario, el acceso global directo a los controles es un error. Los controles pertenecen a las sesiones. Un servicio global no debe guardar para siempre instancias de formularios cualesquiera. En su lugar, las sesiones se suscriben a un servicio, reciben eventos y actualizan sus propios controles en su propio contexto de aplicación.

Un vocabulario seguro para este curso

Sesión significa una instancia en ejecución de una aplicación Wisej.NET para el contexto de una pestaña o ventana del navegador. No es lo mismo que un usuario autenticado: un mismo usuario puede tener varias sesiones.

Contexto de aplicación significa el contexto de Wisej.NET que el código necesita para acceder a las propiedades estáticas de `Application` específicas de la sesión y actualizar el navegador correcto.

Código dentro de la petición (in-bound) significa código que se ejecuta como parte de una petición del navegador o de un evento de Wisej.NET.

Código fuera de la petición (out-of-bound) significa código que se ejecuta fuera de un evento del navegador, como un hilo en segundo plano, un evento de un hosted service, un callback de una cola o un callback de un temporizador.

Push significa que el servidor entrega las actualizaciones de interfaz pendientes al navegador a través de una conexión activa.

Polling significa que el navegador pregunta periódicamente al servidor si hay actualizaciones.



Errores habituales

El error más habitual es actualizar un control desde un hilo en segundo plano sin conservar el contexto de aplicación correcto. El segundo más habitual es hacer push con demasiada frecuencia. Una barra de progreso no necesita 500 actualizaciones por segundo. El push es potente, pero cada actualización tiene un coste: serialización, tráfico de red, trabajo del navegador y planificación en el servidor.

Otro error habitual es usar campos estáticos para datos específicos de la sesión. Los campos estáticos los comparten todos los usuarios y todas las sesiones del dominio de aplicación. Usa `Application.Session`, un servicio por sesión o `Application.GetInstance()` para los objetos específicos de la sesión. Reserva los servicios estáticos para los datos realmente compartidos y, en ese caso, diseña la concurrencia con cuidado.



Punto de control

Al terminar este módulo, el estudiante debe poder dibujar el recorrido completo de un evento de ticket generado en el servidor: el servicio cambia los datos, el servicio lanza un evento, una sesión lo recibe, la sesión actualiza sus controles y Wisej.NET envía los cambios de interfaz resultantes cuando WebSocket está disponible.

Arquitectura web en tiempo real en Wisej.NET

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Crea la primera pantalla de TicketOps Live: una página con una franja de estado en directo, una etiqueta de reloj, un indicador de progreso y un heartbeat simulado del servidor que actualiza el navegador sin que el usuario haga clic.

Punto de partida

Crea una aplicación Wisej.NET Web Page Application llamada `TicketOpsLive`. Añade una página principal llamada `MainPage`. Usa cualquier tema integrado. El laboratorio da por hecho C#, pero los mismos conceptos se aplican a VB.NET.

Tareas

1. Añade un panel superior llamado `statusPanel`.
2. Añade etiquetas llamadas `clockLabel`, `connectionLabel` y `activityLabel`.
3. Añade una barra de progreso llamada `serverLoadBar`.
4. Añade botones llamados `startButton` y `stopButton`.
5. Implementa una tarea en segundo plano de heartbeat simulado al hacer clic en `startButton`.
6. El heartbeat debe actualizar el reloj, un valor de carga ficticio y un mensaje de actividad una vez por segundo.
7. Detén las actualizaciones al hacer clic en `stopButton` o cuando se libere (dispose) la página.

Implementación sugerida

```
private volatile bool _running;

private void startButton_Click(object sender, EventArgs e)
{
    if (_running)
        return;

    _running = true;
    startButton.Enabled = false;
    stopButton.Enabled = true;
    connectionLabel.Text = "Server heartbeat running";

    Application.StartTask(() =>
    {
        var random = new Random();
        var counter = 0;
```

```

while (_running && !this.IsDisposed)
{
    counter++;
    clockLabel.Text = DateTime.Now.ToString("HH:mm:ss");
    serverLoadBar.Value = random.Next(5, 95);
    activityLabel.Text = $"Heartbeat #{counter}";

    Application.Update(this);
    Thread.Sleep(1000);
}
});
}

private void stopButton_Click(object sender, EventArgs e)
{
    _running = false;
    startButton.Enabled = true;
    stopButton.Enabled = false;
    connectionLabel.Text = "Server heartbeat stopped";
}

```

Criterios de aceptación

- ✓ La interfaz se actualiza una vez por segundo después de que el estudiante haga clic en Start.
- ✓ El navegador muestra los cambios sin clics adicionales.
- ✓ Stop desactiva el bucle en menos de un segundo.
- ✓ Pulsar Start dos veces no crea dos bucles que compitan entre sí.
- ✓ El código comprueba si la página se ha liberado (disposed).

Preguntas de reflexión

1. ¿Por qué este laboratorio llama a `Application.Update(this)` dentro del bucle?
2. ¿Qué cambia si WebSocket no está disponible?
3. ¿Por qué `_running` no es un campo estático?
4. ¿Qué riesgo habría en actualizar cada 10 milisegundos?

Contexto de la aplicación, sesiones y ciclo de vida

Sesiones, contexto de aplicación y ciclo de vida: sé dueño de la interfaz del usuario correcto, evita la trampa de los campos estáticos y limpia al salir y al expirar la sesión. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Sesiones, contexto de aplicación y ciclo de vida: sé dueño de la interfaz del usuario correcto, evita la trampa de los campos estáticos y limpia al salir y al expirar la sesión.

Por qué las sesiones importan más en las aplicaciones en tiempo real

En una página tradicional de petición/respuesta, el usuario hace clic, el servidor responde y la mayor parte del trabajo del servidor termina. En una aplicación Wisej.NET en tiempo real, el trabajo puede continuar después de la petición original. Un trabajo en segundo plano puede durar minutos. Un dashboard puede suscribirse a un flujo global de eventos. Un temporizador puede seguir actualizando un gráfico. Como el trabajo sobrevive al clic que lo inició, el ciclo de vida de la sesión pasa a formar parte de la arquitectura de la aplicación.

Una sesión de Wisej.NET se parece a una instancia en ejecución de una aplicación de escritorio: tiene controles, ventanas, páginas, estado de componentes y objetos a nivel de aplicación. Pero, a diferencia de una aplicación de escritorio, el navegador y el servidor pueden desconectarse, el navegador puede recargarse y no existe un hilo de interfaz dedicado y permanente. Wisej.NET restaura el estado de sesión correcto en cada petición y puede mantener el estado del lado del servidor tanto para conexiones HTTP como WebSocket.

Sesión no es lo mismo que usuario

Un usuario autenticado y una sesión están relacionados, pero no son idénticos. Un mismo usuario puede abrir dos pestañas del navegador y crear dos sesiones. Un usuario puede autenticarse dentro de una sesión, cerrar la sesión de usuario y dejar viva la sesión de la aplicación. Un usuario anónimo puede tener una sesión antes de iniciar sesión. Un servicio en segundo plano puede existir sin pertenecer a ninguna sesión de usuario.

Esta distinción es importante para el push. Cuando cambia un ticket, puede que la aplicación tenga que notificar a todas las sesiones de un tenant, a una sola sesión de un operador o a todos los dashboards activos. No des por hecho que «usuario» y «navegador» son lo mismo.

Contexto de aplicación

La clase `Application` de Wisej.NET es estática, pero sus propiedades y operaciones están aisladas por sesión cuando el código se ejecuta en el contexto correcto. Cuando el código se ejecuta porque el navegador ha lanzado un evento, el contexto ya está disponible. Cuando el código se ejecuta en un hilo fuera de la petición, puede que no sepa qué sesión debe actualizar.

`Application.StartTask()` es la forma más sencilla de iniciar trabajo en segundo plano desde una sesión, porque inicia la tarea con el contexto de la sesión que la llama. Cuando el código ya está fuera de la petición, puedes capturar `Application.Current` mientras estás en contexto y usar después `Application.Update(context, action)` o `Application.RunInContext(context, action)` para ejecutar código sobre la sesión correcta.

```
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
}

private void OnExternalEvent(object sender, TicketChangedEventArgs e)
{
    Application.Update(_context, () =>
    {
        notificationsList.Items.Add(e.Message);
    });
}
```

El contexto no es una licencia global para mantener controles vivos para siempre. Debe usarse con una estrategia clara de suscripción y limpieza.

Dónde debe vivir el estado

En una aplicación Wisej.NET en tiempo real hay cuatro lugares habituales para el estado.

Estado de los controles: el estado guardado directamente en los controles, como el texto de una etiqueta, la fila seleccionada de una cuadrícula, la ventana visible, la pestaña actual o el valor de una barra de progreso. Es sencillo y a menudo correcto para el estado de presentación.

Application.Session guarda valores específicos de la sesión que no deben ser visibles para otras sesiones. Úsalo para metadatos ligeros de la sesión, el tenant seleccionado, las preferencias del usuario u objetos por sesión cuando no hace falta un servicio formal.

Servicios de sesión: objetos creados por sesión. Son útiles cuando necesitas un diseño más estructurado y fácil de probar que unos campos de sesión improvisados. Los patrones de Wisej.NET conscientes de la sesión ayudan a evitar compartir datos estáticos por accidente.

Servicios globales: compartidos por todas las sesiones. Son adecuados para colas, hubs de eventos, caché, vigilantes en segundo plano o el estado de todo el sistema. Deben ser thread-safe y no deben guardar controles de sesión sin un ciclo de vida de suscripción bien definido.

La trampa de los campos estáticos

Los campos estáticos se comparten. Si guardas `CurrentTicketId` en un campo estático, cualquier usuario puede sobrescribir el ticket actual de cualquier otro. Si guardas una referencia a `MainPage` en una lista estática y nunca la eliminas, el servidor puede mantener sesiones vivas después de que los usuarios se

vayan. Si guardas un `CancellationTokenSource` en un campo estático para un trabajo de un usuario, un usuario puede cancelar el trabajo de otro.

Usa campos estáticos solo para servicios realmente globales. Para los valores específicos de la sesión, usa controles, `Application.Session` o un objeto por sesión.

Recarga, keep-alive y timeout

Una aplicación Wisej.NET tiene que lidiar con la realidad de la web: los usuarios recargan el navegador, los portátiles entran en suspensión, las redes desaparecen y las pestañas quedan abiertas toda la noche.

Wisej.NET usa el timeout de sesión y el mecanismo de keep-alive para decidir si una sesión debe seguir viva o terminarse. Cuando una sesión expira, se elimina de la memoria. Eso se parece más a cerrar una aplicación de escritorio que a ocultar una página web.

Para las aplicaciones en tiempo real, las implicaciones prácticas son:

- El trabajo en segundo plano de larga duración no debe dar por hecho que la interfaz sigue existiendo.
- El código que actualiza la interfaz debe comprobar si una página o un control se ha liberado (disposed).
- Las suscripciones a eventos globales deben eliminarse cuando la sesión termina o la vista se libera.
- Si un trabajo debe continuar después de que termine una sesión, debe pertenecer a un servicio global, no a la página.
- Si un trabajo solo tiene sentido para la interfaz de un usuario, debe cancelarse cuando esa interfaz desaparezca.

Patrones de limpieza

Una vista en tiempo real que se suscribe a un servicio global siempre debe cancelar la suscripción. El evento concreto depende de la estructura de tu vista, pero el patrón es el mismo: suscribirse en la carga o en la construcción y cancelar la suscripción al liberarse la vista o al salir de la aplicación.

```

private readonly TicketFeed _feed = TicketFeed.Instance;
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
    _feed.TicketChanged += Feed_TicketChanged;
    Application.ApplicationExit += Application_ApplicationExit;
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    _feed.TicketChanged -= Feed_TicketChanged;
}

private void Feed_TicketChanged(object sender, TicketChangedEventArgs e)
{
    if (this.IsDisposed)
        return;

    Application.Update(_context, () =>
    {
        ticketLog.Items.Insert(0, e.Message);
    });
}

```

El objetivo no es solo evitar excepciones, sino hacer evidente quién es dueño de qué. El feed es dueño de los eventos globales de tickets. La página es dueña de sus controles. La sesión es dueña de su contexto de aplicación.

Gestión de errores en las tareas en segundo plano

Una tarea en segundo plano nunca debe fallar en silencio. En una demo de formación, una excepción puede limitarse a detener el bucle. En producción, los fallos silenciosos hacen que los dashboards mientan. Usa un try/catch dentro de la tarea, registra la excepción y actualiza la interfaz con un mensaje de error seguro si la interfaz sigue existiendo.

```
Application.StartTask(() =>
{
    try
    {
        RunLongOperation();
    }
    catch (Exception ex)
    {
        Log.Error(ex);
        if (!this.IsDisposed)
        {
            labelStatus.Text = "The operation failed. See the server log.";
            Application.Update(this);
        }
    }
});
```

No muestres los detalles en bruto de las excepciones a los usuarios finales en producción. Usa registros detallados para los desarrolladores y mensajes seguros para los usuarios.



Punto de control

Al terminar este módulo, el estudiante debe poder explicar por qué una tarea en segundo plano iniciada desde una página puede actualizar esa página, por qué un servicio global no debe ser dueño directo de los controles de una página y por qué toda suscripción en una aplicación en tiempo real necesita un plan para cancelarla.

Contexto de la aplicación, sesiones y ciclo de vida

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Añade un panel de diagnóstico que ayude al desarrollador a ver la sesión actual, el contexto de aplicación, los datos del navegador y los eventos del ciclo de vida relevantes para las actualizaciones en tiempo real.

Tareas

1. Añade un group box de diagnóstico a `MainPage`.
2. Muestra la hora actual, el client id, el tipo de dispositivo del navegador si está disponible y un contador específico de la sesión.
3. Guarda el contador en `Application.Session` o en un objeto por sesión.
4. Añade botones para incrementar el contador y para simular una actualización en segundo plano.
5. Añade un registro de limpieza cuando la aplicación termine.
6. Recarga el navegador y observa qué se conserva.

Fragmentos sugeridos

```
private void MainPage_Load(object sender, EventArgs e)
{
    if (Application.Session.Counter == null)
        Application.Session.Counter = 0;

    clientIdLabel.Text = Application.ClientId;
    titleLabel.Text = "TicketOps Live - " + DateTime.Now.ToShortTimeString();
    Application.Title = "TicketOps Live";

    Application.ApplicationExit += Application_ApplicationExit;
}

private void incrementButton_Click(object sender, EventArgs e)
{
    Application.Session.Counter = (int)Application.Session.Counter + 1;
    counterLabel.Text = Application.Session.Counter.ToString();
}

private void backgroundButton_Click(object sender, EventArgs e)
{
    var context = Application.Current;
    Application.StartTask(() =>
    {
        Thread.Sleep(1500);
        Application.Update(context, () =>
```

```
        {
            lifecycleList.Items.Add("Background update at " + DateTime.Now);
        });
    });
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    // Replace with your logging framework in production.
    System.Diagnostics.Debug.WriteLine("TicketOps session exited: " + Application.ClientId);
}
```

Criterios de aceptación

- ✓ El contador es específico de la sesión, no estático.
- ✓ Una actualización en segundo plano modifica la sesión del navegador correcta.
- ✓ El panel de diagnóstico muestra cuándo llega la actualización en segundo plano.
- ✓ El estudiante sabe explicar qué ocurre después de una recarga y después de que expire la sesión.

Preguntas de reflexión

1. ¿Por qué se captura `Application.Current` antes de iniciar la acción fuera de la petición?
2. ¿Qué pasaría si el contador fuera estático?
3. ¿Qué acción de limpieza añadirías para una suscripción a un servicio global?

Push del servidor para una sesión con tareas en segundo plano

Push para una sola sesión: saca el trabajo largo del manejador del clic, informa del progreso, admite la cancelación cooperativa y restaura la interfaz en finally. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Push para una sola sesión: saca el trabajo largo del manejador del clic, informa del progreso, admite la cancelación cooperativa y restaura la interfaz en finally.

El escenario de push del servidor más habitual

El primer uso más habitual del push del servidor es una operación de larga duración. Un usuario inicia una importación, un informe, un paso de migración, la generación de un archivo, una sincronización de datos o una rutina de validación. La operación tarda segundos o minutos. La interfaz no debe congelarse, el usuario debe ver el progreso y la operación debe informar del éxito o del fallo al terminar.

En las aplicaciones de escritorio, los desarrolladores suelen usar un background worker o una tarea y devolver las actualizaciones al hilo de la interfaz. En Wisej.NET, el diseño equivalente es trabajo en segundo plano en el contexto de la sesión más `Application.Update()` cuando necesitas enviar cambios de interfaz al navegador. La interfaz del lado del servidor sigue perteneciendo a la sesión. La tarea en segundo plano debe respetar el ciclo de vida, la cancelación y la frecuencia de actualización.

Patrón básico

El patrón central tiene seis pasos:

1. Desactiva o ajusta la interfaz antes de empezar el trabajo.
2. Crea el estado de cancelación si el usuario puede cancelar.
3. Inicia la tarea en segundo plano en el contexto de la sesión.
4. Ejecuta el trabajo en pasos pequeños.
5. Actualiza el estado de la interfaz a intervalos razonables.
6. Vuelve a activar la interfaz y muestra la finalización o el fallo.

```

private CancellationTokenSource _cts;

private void startImportButton_Click(object sender, EventArgs e)
{
    _cts = new CancellationTokenSource();
    startImportButton.Enabled = false;
    cancelImportButton.Enabled = true;
    importProgress.Value = 0;
    importStatus.Text = "Starting import ... ";

    Application.StartTask(() => RunImport(_cts.Token));
}

private void cancelImportButton_Click(object sender, EventArgs e)
{
    _cts?.Cancel();
    cancelImportButton.Enabled = false;
    importStatus.Text = "Cancelling ... ";
}

```

El método de la tarea debe contener un try/catch/finally para que la interfaz no quede desactivada después de una excepción.

```

private void RunImport(CancellationToken token)
{
    try
    {
        for (int i = 1; i ≤ 100; i++)
        {
            token.ThrowIfCancellationRequested();

            // Simulate work.
            Thread.Sleep(75);

            importProgress.Value = i;
            importStatus.Text = $"Imported {i}%";

            if (i % 5 == 0)
                Application.Update(this);
        }

        importStatus.Text = "Import complete.";
    }
    catch (OperationCanceledException)
    {
        importStatus.Text = "Import cancelled.";
    }
    catch (Exception ex)
    {
        LogError(ex);
        importStatus.Text = "Import failed. See the server log.";
    }
}

```

```

finally
{
    startImportButton.Enabled = true;
    cancelImportButton.Enabled = false;
    Application.Update(this);
}
}

```

Fíjate en la frecuencia de actualización. El código cambia la barra de progreso en cada iteración, pero solo hace push cada cinco iteraciones, más la actualización final. Así se reduce el número de mensajes de red y de repintados del navegador sin dejar de transmitir la sensación de directo.

Actualización directa de la interfaz frente a callback con contexto

Cuando una tarea se inicia con `Application.StartTask()` desde un evento de Wisej.NET, se ejecuta con el contexto de aplicación actual. En muchos ejemplos verás cambios directos en los controles seguidos de `Application.Update(this)`. Es aceptable para un trabajo de una sola sesión cuyo dueño es la página.

Para el código que se inicia fuera de la sesión, usa la sobrecarga con contexto explícito:

```

var context = Application.Current;

ExternalWorker.JobAdvanced += (s, e) =>
{
    Application.Update(context, () =>
    {
        progressBar.Value = e.Percent;
        labelStatus.Text = e.Message;
    });
};

```

Así queda claro que los cambios de interfaz ocurren en el contexto de sesión correcto.

No bloquee los manejadores de eventos

No ejecutes una operación larga directamente en un manejador de clic. La petición del navegador queda abierta, el usuario no ve actualizaciones intermedias por push y el hilo del servidor queda ocupado demasiado tiempo. Inicia la operación y devuelve el control al usuario.

Mal:

```

private void button_Click(object sender, EventArgs e)
{
    RunTenMinuteImport();
    labelStatus.Text = "Done";
}

```

Mejor:

```
private void button_Click(object sender, EventArgs e)
{
    Application.StartTask() => RunTenMinuteImportWithProgress();
    labelStatus.Text = "Import started";
}
```

La cancelación es una funcionalidad, no un añadido de última hora

Una interfaz de progreso en tiempo real sin cancelación puede ser peor que una interfaz bloqueante: le dice al usuario que algo está pasando, pero no le da ningún control. La cancelación debe integrarse en las tareas de larga duración siempre que sea seguro. No termines hilos a la fuerza. Usa la cancelación cooperativa. Comprueba el token entre unidades de trabajo y deja el sistema en un estado coherente.

La cancelación también es útil cuando se cierra la página. Si el trabajo en segundo plano es solo para esa página, cáncélalo al liberarla o al salir de la aplicación. Si el trabajo es un proceso del servidor que debe continuar, traslada su propiedad a un servicio y deja que la página muestre el estado del servicio.

Notificación de errores

Una tarea en segundo plano puede fallar después de que el manejador de eventos original haya terminado. Sin una gestión de errores explícita, puede que el usuario solo vea que el progreso se ha detenido. Captura siempre las excepciones dentro del trabajo en segundo plano. Registra los detalles técnicos. Envía a la interfaz un estado de error seguro para el usuario. Conserva el detalle suficiente para poder diagnosticar: id del trabajo, id de sesión o de cliente, id de usuario si está autenticado, nombre de la operación y hora.

Agrupación y limitación de la frecuencia

El ojo humano no se beneficia de actualizaciones ilimitadas. La mayoría de los dashboards de negocio parecen en directo con 1-4 actualizaciones por segundo. Las barras de progreso suelen verse fluidas con 5-10 actualizaciones por segundo, pero muchas operaciones no lo necesitan. La telemetría de alta frecuencia debe agregarse antes de mostrarse.

Reglas prácticas:

- Envía el estado final de inmediato.
- Envía el progreso visible para el usuario a un intervalo controlado.
- Agrupa varios cambios de etiquetas o cuadrículas en una sola llamada a `Application.Update()`.
- Evita hacer push dentro de bucles intensivos de CPU.
- Prefiere una cola o un temporizador para agrupar las ráfagas que llegan de sistemas externos.



Punto de control

Al terminar este módulo, el estudiante debe poder implementar una operación de larga duración que se inicie de inmediato, actualice el progreso sin bloquear el navegador, admita la cancelación, informe de los errores y deje la interfaz en un estado coherente.

Push del servidor para una sesión con tareas en segundo plano

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Añade a TicketOps Live un panel de importación de larga duración realista. El usuario puede iniciar una importación, seguir su progreso, cancelarla y examinar el resultado.

Requisitos de la interfaz

Añade:

- ✓ `startImportButton`
- ✓ `cancelImportButton`
- ✓ `importProgressBar`
- ✓ `importStatusLabel`
- ✓ `importLogListBox`
- ✓ `recordsImportedLabel`

Tareas

1. Implementa un bucle de importación con 200 registros simulados.
2. Envía el progreso cada 10 registros, no en cada registro.
3. Añade mensajes de log para el inicio, cada 50 registros, la cancelación, el fallo y la finalización.
4. Implementa la cancelación con `CancellationTokenSource`.
5. Añade un botón o una opción de fallo deliberado que lance una excepción en el registro 87.
6. Asegúrate de que la interfaz se vuelve a activar en todos los desenlaces.

Bucle de registros sugerido

```
private void RunImport(CancellationToken token, bool failAt87)
{
    try
    {
        for (int i = 1; i ≤ 200; i++)
        {
            token.ThrowIfCancellationRequested();
            Thread.Sleep(25);

            if (failAt87 && i == 87)
                throw new InvalidOperationException("Simulated malformed record.");

            recordsImportedLabel.Text = i.ToString();
            importProgressBar.Value = i / 2;
        }
    }
}
```

```

        if (i % 50 == 0)
            importLogListBox.Items.Insert(0, $"Imported {i} records");

        if (i % 10 == 0)
            Application.Update(this);
    }

    importStatusLabel.Text = "Import completed successfully.";
}
catch (OperationCanceledException)
{
    importStatusLabel.Text = "Import cancelled by user.";
    importLogListBox.Items.Insert(0, "Cancelled at " + DateTime.Now);
}
catch (Exception ex)
{
    LogError(ex);
    importStatusLabel.Text = "Import failed. Review the server log.";
    importLogListBox.Items.Insert(0, "Failed at " + DateTime.Now);
}
finally
{
    startImportButton.Enabled = true;
    cancelImportButton.Enabled = false;
    Application.Update(this);
}
}

```

Criterios de aceptación

- ✓ Las actualizaciones de progreso aparecen mientras la operación está en curso.
- ✓ La cancelación funciona y deja la interfaz utilizable.
- ✓ El fallo se notifica sin exponer stack traces en bruto.
- ✓ La actualización final siempre llega al navegador cuando la sesión sigue viva.
- ✓ El código no hace más push del necesario.

Reto adicional

Añade un `JobId` a cada importación e inclúyelo en los mensajes de log. El proyecto final lo usará para relacionar los eventos de la interfaz con los logs del servidor.

Temporizadores, sondeo de respaldo y cadencia de actualización

Cadencia de actualización y alternativas: elige entre push, temporizadores y polling, agrupa las ráfagas y gestiona el ciclo de vida de los temporizadores por sesión. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Cadencia de actualización y alternativas: elige entre push, temporizadores y polling, agrupa las ráfagas y gestiona el ciclo de vida de los temporizadores por sesión.

El push es preferible, pero la alternativa importa

Una aplicación en producción no puede dar por hecho que todos los navegadores, proxies, servidores o rutas de red se comportarán a la perfección. Algunos entornos bloquean los WebSockets. Algunos proxies inversos necesitan que se habilite explícitamente el soporte de WebSocket. Algunas redes corporativas introducen timeouts. Wisej.NET admite push en tiempo real con WebSocket y también ofrece alternativas, como el polling, para los sistemas que no pueden usar WebSocket.

El desarrollador de nivel intermedio debe entender la diferencia entre una alternativa y un error de diseño. El polling no es incorrecto por definición. Es incorrecto cuando se usa como sustituto de un modelo de eventos claro o cuando el intervalo se elige sin tener en cuenta el coste para el servidor. Un intervalo de polling de un segundo para miles de sesiones puede salir caro. Un intervalo de diez segundos para un tablero de operaciones crítico puede resultar lento. La elección correcta depende del valor para el usuario y del coste.

Wisej.Web.Timer frente a tarea en segundo plano

Un `Wisej.Web.Timer` es útil cuando quieres eventos periódicos del lado del servidor asociados a una sesión. Por ejemplo, un dashboard puede recalcular un KPI cada cinco segundos aunque no haya llegado ningún evento de un servicio global. Una tarea en segundo plano es mejor cuando una operación concreta empieza, se ejecuta y termina. Un servicio global es mejor cuando un único vigilante compartido alimenta muchas sesiones.

Usa un temporizador para el refresco periódico de la interfaz con ámbito de sesión. Usa `Application.StartTask()` para el trabajo de larga duración con ámbito de sesión. Usa un servicio compartido para el estado global y los eventos multiusuario.

Polling como alternativa con StartPolling y EndPolling

Cuando WebSocket está disponible, `Application.Update()` puede enviar las actualizaciones pendientes al navegador. Cuando WebSocket no está disponible, no tiene ninguna conexión activa que usar.

Wisej.NET ofrece `Application.StartPolling(milliseconds)` y `Application.EndPolling()` como mecanismos alternativos. El polling se ignora cuando WebSocket está activo, lo que permite a la aplicación solicitar el comportamiento alternativo sin perjudicar a las sesiones WebSocket.

Un patrón habitual es:

```
private void BeginLiveMode()
{
    Application.StartPolling(1000);

    Application.StartTask(() =>
    {
        while (_liveMode && !this.IsDisposed)
        {
            UpdateDashboardModel();
            ApplyDashboardToControls();

            // Effective when WebSocket exists. Polling handles non-WebSocket cases.
            Application.Update(this);
            Thread.Sleep(1000);
        }

        Application.EndPolling();
    });
}
```

Aun así, el código debe diseñarse con cuidado. Si `_liveMode` nunca pasa a false y la página se libera, el bucle se convierte en una fuga. Ten siempre una condición de parada.

Elegir la cadencia de actualización

La cadencia es el ritmo de las actualizaciones. En una aplicación Wisej.NET en tiempo real, la cadencia tiene cuatro dimensiones:

Cadencia del modelo: con qué frecuencia cambian los datos subyacentes.

Cadencia de la interfaz: con qué frecuencia el usuario necesita ver cambios visibles.

Cadencia del push: con qué frecuencia el servidor envía actualizaciones de interfaz al navegador.

Cadencia del polling: con qué frecuencia el navegador pide actualizaciones al servidor cuando el push no está disponible.

No siempre coinciden. Una máquina puede producir telemetría 100 veces por segundo, pero puede que la interfaz solo necesite un resumen una vez por segundo. Una cola puede cambiar 20 veces en una ráfaga, pero el dashboard puede mostrar el estado final pasados 250 milisegundos.

Agrupar las ráfagas

Una ráfaga se produce cuando llegan muchos cambios a la vez. Si cada cambio provoca un push independiente, la interfaz puede desperdiciar recursos y mostrar estados intermedios que los usuarios no entienden. Agrupar (coalescing) significa reunir varios cambios y aplicarlos juntos.

Estrategia de agrupación sencilla:

```

private readonly object _sync = new object();
private bool _dirty;

private void OnModelChanged()
{
    lock (_sync)
    {
        _dirty = true;
    }
}

private void refreshTimer_Tick(object sender, EventArgs e)
{
    bool shouldRefresh;

    lock (_sync)
    {
        shouldRefresh = _dirty;
        _dirty = false;
    }

    if (!shouldRefresh)
        return;

    ApplyModelToControls();
}

```

El evento marca el modelo como modificado (dirty). El temporizador aplica las actualizaciones a una cadencia de interfaz estable.

La propiedad del temporizador

Todo temporizador pertenece a alguien. Si es un componente de Wisej.NET en una página, su dueño es la página. Si es un temporizador de .NET en un servicio global, su dueño es el servicio. La propiedad determina la limpieza. Un temporizador duplicado puede duplicar el tráfico de actualizaciones y hacer que los errores parezcan aleatorios.

Reglas prácticas:

- Inicia los temporizadores una sola vez.
- Detén los temporizadores cuando la vista deje de estar activa.
- No crees un temporizador nuevo en cada clic del botón de refresco.
- Evita el trabajo de larga duración dentro de un tick del temporizador; en su lugar, inicia o planifica ese trabajo.
- Usa un indicador para evitar ticks solapados si un refresco tarda más que el intervalo.

Detectar una mala cadencia

Los síntomas de demasiadas actualizaciones incluyen un uso elevado de CPU, un renderizado entrecortado en el navegador, mucho tráfico WebSocket, ruido en los logs del servidor y dashboards que parecen

parpadear. Los síntomas de muy pocas actualizaciones incluyen usuarios que pulsán refrescar, trabajo duplicado, decisiones operativas basadas en datos obsoletos y falta de confianza en el dashboard.

Los desarrolladores de nivel intermedio deben instrumentar los dashboards importantes. Cuenta los eventos recibidos, las actualizaciones de interfaz aplicadas, los push intentados, los errores y el tiempo medio empleado en aplicar una actualización.



Punto de control

Al terminar este módulo, el estudiante debe poder elegir entre un temporizador, una tarea en segundo plano, el polling como alternativa o un evento de servicio según el trabajo que haya que hacer, y explicar el coste de la cadencia elegida.

Temporizadores, sondeo de respaldo y cadencia de actualización

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Amplía TicketOps Live con un temporizador de refresco del dashboard y controles para experimentar con la cadencia de actualización.

Requisitos de la interfaz

Añade:

- ✓ `liveModeCheckBox`
- ✓ `cadenceComboBox` con los valores `250 ms`, `1 sec`, `5 sec`
- ✓ `eventsReceivedLabel`
- ✓ `updatesAppliedLabel`
- ✓ `lastAppliedLabel`
- ✓ `refreshTimer`

Tareas

1. Inicia el polling cuando se active el modo en directo.
2. Detén el polling cuando se desactive el modo en directo.
3. Usa un temporizador para aplicar las actualizaciones del dashboard a la cadencia seleccionada.
4. Simula cambios del modelo más rápidos que las actualizaciones de la interfaz.
5. Cuenta por separado los eventos del modelo y las actualizaciones de interfaz aplicadas.
6. Evita que se solapen los ticks del temporizador.

Idea de implementación sugerida

```
private int _eventsReceived;
private int _updatesApplied;
private bool _refreshInProgress;
private volatile bool _dashboardDirty;

private void liveModeCheckBox_CheckedChanged(object sender, EventArgs e)
{
    if (liveModeCheckBox.Checked)
    {
        Application.StartPolling(1000);
        refreshTimer.Interval = GetSelectedInterval();
        refreshTimer.Start();
    }
    else
```

```

    {
        refreshTimer.Stop();
        Application.EndPolling();
    }
}

private void SimulatedModelChanged()
{
    _eventsReceived++;
    _dashboardDirty = true;
}

private void refreshTimer_Tick(object sender, EventArgs e)
{
    if (_refreshInProgress || !_dashboardDirty)
        return;

    try
    {
        _refreshInProgress = true;
        _dashboardDirty = false;
        _updatesApplied++;

        eventsReceivedLabel.Text = _eventsReceived.ToString();
        updatesAppliedLabel.Text = _updatesApplied.ToString();
        lastAppliedLabel.Text = DateTime.Now.ToString("HH:mm:ss.fff");
    }
    finally
    {
        _refreshInProgress = false;
    }
}

```

Criterios de aceptación

- ✓ El cambio de cadencia del temporizador es visible.
- ✓ El contador de eventos recibidos puede crecer más rápido que el de actualizaciones de interfaz aplicadas.
- ✓ El polling se inicia y se detiene con el modo en directo.
- ✓ No se producen ticks del temporizador solapados.
- ✓ El estudiante sabe explicar por qué es útil agrupar las actualizaciones.

Enlace de datos en vivo y cuadrículas en tiempo real

Enlace de datos en directo: actualiza las colecciones enlazadas en el contexto de la sesión sin volver a enlazar la cuadrícula, conservando la selección, el orden y el desplazamiento. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Enlace de datos en directo: actualiza las colecciones enlazadas en el contexto de la sesión sin volver a enlazar la cuadrícula, conservando la selección, el orden y el desplazamiento.

Por qué el enlace de datos se complica cuando los datos se mueven

El enlace de datos estático es sencillo: cargar registros, enlazar la cuadrícula, dejar que el usuario edite y guardar. El enlace de datos en tiempo real añade movimiento. Llegan filas, cambian campos, avanzan estados y los trabajos en segundo plano actualizan la misma pantalla que el usuario está mirando. El reto es mantener la interfaz sincronizada sin destruir y recrear los enlaces constantemente.

Wisej.NET admite el enlace simple para propiedades de un solo control y el enlace complejo para controles como las cuadrículas. El componente `BindingSource` es especialmente útil porque se sitúa entre los controles y los orígenes de datos, gestiona el elemento actual (currency) y ofrece un punto coherente para restablecer o refrescar los enlaces.

El modelo Ticket

En este curso se usa un modelo de ticket sencillo:

```
public enum TicketStatus
{
    New,
    Assigned,
    Waiting,
    Resolved,
    Escalated
}

public class Ticket : INotifyPropertyChanged
{
    private TicketStatus _status;
    private string _owner;
    private DateTime _updatedAt;

    public int Id { get; set; }
    public string Title { get; set; }
    public string Customer { get; set; }

    public string Owner
    {
        get => _owner;
    }
}
```

```

        set { _owner = value; OnPropertyChanged(nameof(Owner)); }
    }

    public TicketStatus Status
    {
        get => _status;
        set { _status = value; OnPropertyChanged(nameof(Status)); }
    }

    public DateTime UpdatedAt
    {
        get => _updatedAt;
        set { _updatedAt = value; OnPropertyChanged(nameof(UpdatedAt)); }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

La finalidad de `INotifyPropertyChanged` es que los cambios de propiedades sean visibles para la infraestructura de enlace allí donde esté soportado. Aunque llames a `BindingSource.ResetBindings(false)` para mayor claridad, el modelo sigue expresando que puede notificar cambios.

Enlazar una lista en directo

Un punto de partida práctico es `BindingList<Ticket>` junto con `BindingSource`.

```

private BindingList<Ticket> _tickets;
private BindingSource _ticketSource;

private void MainPage_Load(object sender, EventArgs e)
{
    _tickets = new BindingList<Ticket>();
    _ticketSource = new BindingSource();
    _ticketSource.DataSource = _tickets;

    ticketsGrid.AutoGenerateColumns = true;
    ticketsGrid.DataSource = _ticketSource;
}

```

Cuando llega un ticket nuevo, añádelo a la lista en el contexto de sesión correcto. Cuando cambia un campo, actualiza el elemento y restablece los enlaces si la interfaz no se refresca automáticamente.

```
private void AddTicket(Ticket ticket)
{
    _tickets.Insert(0, ticket);
    _ticketSource.ResetBindings(false);
}

private void UpdateTicket(int id, TicketStatus status, string owner)
{
    var ticket = _tickets.FirstOrDefault(t => t.Id == id);
    if (ticket == null)
        return;

    ticket.Status = status;
    ticket.Owner = owner;
    ticket.UpdatedAt = DateTime.Now;
    _ticketSource.ResetBindings(false);
}
```

No vuelvas a enlazar la cuadrícula en cada actualización

Volver a enlazar toda la cuadrícula es a veces la forma más rápida de que una demo funcione, pero en producción suele ser un error. Puede restablecer la selección, la posición de desplazamiento, el orden, el ancho de las columnas y el contexto del usuario. Además, puede ser costoso. Prefiere los cambios incrementales en la lista enlazada y después notifica al binding source.

Mal:

```
ticketsGrid.DataSource = null;
ticketsGrid.DataSource = LoadAllTicketsAgain();
```

Mejor:

```
ApplyTicketChange(change);
_ticketSource.ResetBindings(false);
```

Lo mejor para pantallas con mucho volumen: agrupa varios cambios y refresca una vez por intervalo de cadencia.

Seguridad entre hilos con colecciones enlazadas

Una colección enlazada pertenece a la interfaz de la sesión. Un servicio global puede recibir actualizaciones desde otro hilo. No modifiques la colección enlazada desde hilos cualesquiera sin entrar en el contexto de la sesión y sin tener en cuenta la sincronización.

Un patrón multiusuario seguro es:

1. El servicio global recibe un evento de ticket.
2. El manejador de eventos de la sesión captura el evento.
3. El manejador usa `Application.Update(context, () => { ... })`.

4. Dentro del callback, actualiza la lista enlazada de la sesión.
5. Restablece los enlaces una sola vez.

```
private void Feed_TicketChanged(object sender, TicketChangedEventArgs e)
{
    Application.Update(_context, () =>
    {
        ApplyTicketChange(e.Change);
        _ticketSource.ResetBindings(false);
    });
}
```

Formato y análisis de valores

Las cuadrículas en tiempo real no consisten solo en transportar datos: deben seguir siendo legibles. Aplica formato a las fechas, los importes, las prioridades y los estados. Usa los eventos de parse/format de los controles enlazados cuando los valores requieran conversión. Evita guardar cadenas ya formateadas en el modelo si el valor real tiene un tipo más adecuado.

Por ejemplo, `UpdatedAt` debe ser un `DateTime`, no una cadena preformateada. La cuadrícula puede mostrarlo como hora local o como antigüedad relativa.

Estado de las filas y señales visuales

Una cuadrícula en directo debe mostrar qué ha cambiado sin volverse ruidosa. Patrones habituales:

- mostrar un indicador temporal de «actualizado» durante unos segundos;
- ordenar los tickets nuevos en la parte superior;
- mantener estable el ticket seleccionado por el usuario siempre que sea posible;
- evitar mover una fila mientras el usuario la está editando;
- usar columnas de estado e indicadores pequeños en lugar de interrupciones modales.

La interfaz debe ayudar a los usuarios a confiar en los datos. Demasiadas animaciones o saltos de filas pueden hacerles pensar que la aplicación es inestable.

Validación y gestión de conflictos

Datos en tiempo real significa que otro usuario o servicio puede cambiar un registro mientras el usuario actual lo está viendo o editando. Decide qué ocurre cuando un ticket cambia durante la edición. En este curso, usa una regla sencilla: si un ticket se está editando en un formulario de detalle, muestra un aviso no bloqueante de que el ticket ha cambiado y permite al usuario recargarlo o continuar.

Los sistemas de producción pueden necesitar concurrencia optimista con números de versión, marcas de tiempo o row versions de la base de datos. Wisej.NET no elimina la necesidad de concurrencia en la capa de datos: le da a la interfaz una forma de avisar a los usuarios en el momento adecuado.



Punto de control

Al terminar este módulo, el estudiante debe poder mantener una cuadrícula en directo que recibe filas nuevas y actualizaciones de filas desde un origen en segundo plano, conservando el contexto del usuario y evitando volver a enlazar sin necesidad.

Enlace de datos en vivo y cuadrículas en tiempo real

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Crea un `DataGridView` en directo que muestre tickets de soporte y se actualice cuando lleguen eventos de ticket simulados.

Tareas

1. Añade un `DataGridView` llamado `ticketsGrid`.
2. Crea el modelo `Ticket` y el enum `TicketStatus`.
3. Enlaza una `BindingList<Ticket>` a través de un `BindingSource`.
4. Añade un botón que cree un evento de ticket nuevo cada segundo durante 20 segundos.
5. Añade un botón que cambie aleatoriamente el estado de los tickets.
6. Conserva la fila seleccionada siempre que sea posible.
7. Muestra `UpdatedAt` en un formato legible.

Evento del modelo sugerido

```
public class TicketChangedEventArgs : EventArgs
{
    public Ticket Ticket { get; set; }
    public string ChangeType { get; set; }
    public string Message { get; set; }
}
```

Método de actualización sugerido

```
private void ApplyTicketEvent(TicketChangedEventArgs e)
{
    var selectedId = ticketsGrid.CurrentRow?.DataBoundItem is Ticket selected
        ? selected.Id
        : (int?)null;

    var existing = _tickets.FirstOrDefault(t => t.Id == e.Ticket.Id);

    if (existing == null)
        _tickets.Insert(0, e.Ticket);
    else
    {
        existing.Status = e.Ticket.Status;
        existing.Owner = e.Ticket.Owner;
        existing.UpdatedAt = DateTime.Now;
    }

    _ticketSource.ResetBindings(false);

    if (selectedId.HasValue)
        RestoreSelection(selectedId.Value);
}
```

Criterios de aceptación

- ✓ Los tickets nuevos aparecen sin volver a enlazar la cuadrícula desde cero.
- ✓ Los tickets existentes se actualizan en su sitio.
- ✓ La fila seleccionada se conserva cuando la fila sigue existiendo.
- ✓ La frecuencia de actualización se mantiene bajo control.
- ✓ El estudiante sabe explicar por qué la lista enlazada pertenece a la sesión.

Reto adicional

Añade un filtro para los tickets **Escalated** y asegúrate de que las actualizaciones en directo siguen funcionando con el filtro activo.

Push multiusuario con servicios y hubs de eventos

Push multiusuario: un hub de eventos global lanza eventos de dominio mientras cada sesión actualiza su propia interfaz en su propio contexto. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Push multiusuario: un hub de eventos global lanza eventos de dominio mientras cada sesión actualiza su propia interfaz en su propio contexto.

De una sesión a muchas sesiones

El push para una sola sesión es útil, pero muchas aplicaciones de negocio necesitan push multiusuario. Cuando un despachador asigna un ticket, todos los dashboards de los supervisores deben actualizarse. Cuando termina una importación nocturna, todos los administradores conectados deben ver el resultado. Cuando falla una comprobación de salud de un servicio, las consolas de operaciones activas deben mostrar la alerta.

El cambio arquitectónico clave es la propiedad. Una página es dueña de sus controles. Una sesión es dueña de su contexto. Un servicio global es dueño de los datos y los eventos compartidos. No permitas que el servicio global se convierta en un saco de referencias a formularios. En su lugar, usa eventos de dominio y suscripciones a nivel de sesión.

El patrón event hub

Un hub de eventos es un servicio compartido que recibe o crea eventos de dominio y los expone a los suscriptores. El hub no sabe cómo muestra el evento cada interfaz: simplemente anuncia que algo ha ocurrido.

```
public sealed class TicketHub
{
    private static readonly Lazy<TicketHub> _instance =
        new Lazy<TicketHub>(() => new TicketHub());

    public static TicketHub Instance => _instance.Value;

    public event EventHandler<TicketChangedEventArgs> TicketChanged;

    private readonly object _sync = new object();
    private readonly List<Ticket> _tickets = new List<Ticket>();

    private TicketHub() { }

    public IReadOnlyList<Ticket> GetSnapshot()
    {
        lock (_sync)
            return _tickets.Select(CloneTicket).ToList();
    }
}
```

```

public void AddOrUpdate(Ticket ticket)
{
    TicketChangedEventArgs args;

    lock (_sync)
    {
        var existing = _tickets.FirstOrDefault(t => t.Id == ticket.Id);
        if (existing == null)
            _tickets.Insert(0, ticket);
        else
            CopyTicket(ticket, existing);

        args = new TicketChangedEventArgs
        {
            Ticket = CloneTicket(ticket),
            ChangeType = existing == null ? "Added" : "Updated",
            Message = $"Ticket {ticket.Id} {ticket.Status}"
        };
    }

    TicketChanged?.Invoke(this, args);
}
}

```

El hub mantiene el estado global de forma thread-safe y lanza un evento de dominio. No llama a `Application.Update()` porque no es dueño de ninguna sesión.

El patrón de suscriptor de sesión

Cada sesión se suscribe al hub y actualiza su propia interfaz.

```

private readonly TicketHub _hub = TicketHub.Instance;
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
    LoadSnapshot(_hub.GetSnapshot());
    _hub.TicketChanged += Hub_TicketChanged;
    Application.ApplicationExit += Application_ApplicationExit;
}

private void Hub_TicketChanged(object sender, TicketChangedEventArgs e)
{
    if (this.IsDisposed)
        return;

    Application.Update(_context, () =>
    {
        ApplyTicketEvent(e);
        notificationsList.Items.Insert(0, e.Message);
    });
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    _hub.TicketChanged -= Hub_TicketChanged;
}

```

Este patrón da a cada sesión su propia interfaz. Dos usuarios pueden filtrar u ordenar de forma distinta mientras reciben los mismos eventos de dominio.

Difusión frente a actualización dirigida

Difusión (broadcast) significa que todos los suscriptores reciben el evento. Actualización dirigida significa que solo lo reciben algunas sesiones. Ejemplos:

- Difundir todas las alertas graves del sistema a todas las consolas de operaciones.
- Enviar los eventos específicos de un tenant solo a las sesiones de ese tenant.
- Enviar las notificaciones de asignación solo al operador asignado.
- Enviar los eventos de administración solo a los usuarios con rol de administrador.

El hub puede incluir metadatos suficientes para que el suscriptor decida si muestra el evento. Otra opción es que el hub gestione grupos de suscripción. Mantén sencilla la primera versión: incluye metadatos de tenant, usuario o rol y deja que la sesión filtre.

Seguridad entre hilos y snapshots

Los servicios globales suelen mantener colecciones. Esas colecciones deben protegerse con locks o usar colecciones concurrentes. Al exponer datos a las sesiones, prefiere los snapshots a las referencias directas

a la colección. Un snapshot evita que el código de interfaz modifique el estado global por accidente y evita recorrer la colección mientras el servicio la modifica.

Mal:

```
public List<Ticket> Tickets => _tickets;
```

Mejor:

```
public IReadOnlyList<Ticket> GetSnapshot()
{
    lock (_sync)
        return _tickets.Select(CloneTicket).ToList();
}
```

La copia no tiene por qué ser costosa en todos los casos de uso, pero el límite de propiedad debe estar claro.

Fugas de memoria a través de eventos

Los eventos son una fuente habitual de fugas de memoria. Si un servicio global tiene un evento y una página se suscribe, el servicio pasa a tener una referencia al manejador de la página. Si la página se cierra pero nunca cancela la suscripción, el servicio puede mantenerla viva. En las aplicaciones en tiempo real, esta fuga es a la vez un problema de memoria y de corrección, porque el servicio puede intentar actualizar una sesión muerta.

Toda suscripción debe tener su cancelación. Es una regla del curso.

Backpressure

Si el hub emite cambios más rápido de lo que las sesiones pueden renderizarlos, el sistema necesita backpressure o agrupación. Algunas estrategias sencillas:

- descartar los eventos duplicados de baja prioridad;
- agrupar los cambios y enviar resúmenes;
- marcar una sesión como modificada y refrescarla con un temporizador;
- conservar solo el último estado de una métrica del dashboard;
- encolar los eventos por sesión con un tamaño máximo.

Las aplicaciones de negocio no suelen necesitar renderizar cada estado intermedio. Necesitan renderizar el estado sobre el que los usuarios pueden actuar.



Punto de control

Al terminar este módulo, el estudiante debe poder diseñar un servicio compartido que alimente de forma segura muchas sesiones de Wisej.NET sin guardar controles de página a nivel global, sin fugas de manejadores de eventos y sin modificar colecciones de interfaz enlazadas desde el hilo equivocado.

Push multiusuario con servicios y hubs de eventos

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Refactoriza la simulación de tickets en un servicio global `TicketHub` y haz que varias sesiones del navegador reciban eventos de ticket en directo.

Tareas

1. Crea un singleton `TicketHub` con una lista de tickets thread-safe.
2. Añade eventos `TicketChanged`.
3. Carga un snapshot cuando se inicie `MainPage`.
4. Suscribe la página a `TicketChanged`.
5. Cancela la suscripción al salir de la aplicación.
6. Abre dos navegadores o dos sesiones y comprueba que ambos reciben las actualizaciones.
7. Añade metadatos de tenant o de rol y filtra al menos un tipo de evento.

Criterios de aceptación

- ✓ El hub no guarda referencias a páginas ni a controles.
- ✓ Cada sesión actualiza su propia lista enlazada.
- ✓ Cerrar o recargar una sesión no deja atrás un suscriptor roto.
- ✓ Varias sesiones pueden mostrar filtros distintos compartiendo el mismo origen de eventos de dominio.
- ✓ El estudiante sabe explicar la diferencia entre difusión y actualización dirigida.

Reto adicional

Añade un contador de notificaciones por sesión. El contador debe ser específico de la sesión aunque los eventos procedan de un servicio global.

Refuerzo para producción, despliegue y proyecto final

Refuerzo para producción: verifica toda la ruta WebSocket, las sticky sessions, los health checks, la seguridad y la observabilidad antes del despliegue. El concepto, el modelo mental y los errores habituales en producción: léelo antes del laboratorio.



Enfoque de producción

Refuerzo para producción: verifica toda la ruta WebSocket, las sticky sessions, los health checks, la seguridad y la observabilidad antes del despliegue.

El tiempo real en producción es arquitectura más operaciones

Una aplicación en tiempo real puede funcionar perfectamente en la máquina del desarrollador y fallar en producción porque la ruta de despliegue es distinta. Un proxy puede bloquear WebSocket. Un balanceador de carga puede enviar las peticiones de un usuario a instancias de servidor distintas. Un timeout de sesión puede terminar el trabajo de interfaz de larga duración. Un servicio global puede consumir demasiada memoria. Un ajuste de seguridad puede ser seguro en desarrollo pero incorrecto detrás de un proxy inverso.

Este módulo convierte los patrones del curso en una lista de comprobación para producción.

Un despliegue preparado para WebSocket

El push en tiempo real de Wisej.NET funciona mejor cuando WebSocket está disponible. En producción, verifica el soporte de WebSocket en toda la ruta: navegador, red, proxy inverso, balanceador de carga, servidor web y aplicación. Algunos balanceadores de carga necesitan enrutamiento TCP o soporte explícito de WebSocket. Los proxies inversos como NGINX pueden hacer de proxy para WebSocket, pero deben estar bien configurados.

Probar solo desde la consola del servidor no basta. Prueba desde la ruta de red del usuario. Una VPN corporativa, un dispositivo de seguridad o un proxy antiguo pueden cambiar el comportamiento.

Balanceo de carga y afinidad de sesión

Las aplicaciones Wisej.NET mantienen el estado de la sesión en el servidor. En un despliegue con varios servidores, las peticiones de una sesión dada deben llegar siempre a la misma instancia de servidor, salvo que la arquitectura admita expresamente otra estrategia. La regla práctica de balanceo de carga es usar sticky sessions u otro mecanismo de enrutamiento que garantice la coherencia entre usuario y servidor.

Los health checks deben indicar al balanceador de carga cuándo un servidor debe dejar de recibir tráfico. Wisej.NET admite la configuración de health checks mediante un archivo `HealthCheck.json`, con parámetros como la memoria máxima, el número máximo de sesiones y el umbral de CPU. Los umbrales exactos dependen del comportamiento de la aplicación y del tamaño del hosting.

Áreas de configuración importantes

La configuración está repartida entre `Web.config` y los archivos de configuración JSON de Wisej.NET. Las aplicaciones Wisej.NET usan archivos de aplicación JSON como `Default.json`, y un proyecto puede definir varias aplicaciones Wisej.NET, cada una con sus propios archivos de configuración.

Entre los ajustes de tiempo real y las cuestiones de despliegue importantes están:

- `sessionTimeout` : controla cuándo expiran las sesiones inactivas;
- `pollingInterval` : controla el comportamiento del polling automático para los clientes sin WebSocket;
- `responseTimeout` : importa para las respuestas HTTP iniciales largas, pero WebSocket no expira de la misma manera;
- `debug=false` en producción, para que las bibliotecas JavaScript se empaqueten y minifiquen correctamente;
- la clave de licencia del servidor y la configuración del tema predeterminado;
- la configuración de módulos y handlers de IIS para las aplicaciones .NET Framework;
- el modelo de hosting y el middleware de ASP.NET Core para las aplicaciones .NET Core/.NET;
- la configuración de health checks para los despliegues con balanceo de carga.

Fragmento de ejemplo de `Default.json` para una aplicación de formación:

```
{
  "mainWindow": "TicketOpsLive.MainPage, TicketOpsLive",
  "theme": "Bootstrap-4",
  "sessionTimeout": 180,
  "pollingInterval": 1000
}
```

No copies a ciegas los valores de formación en producción. Usa pruebas de carga y los requisitos de los usuarios para elegir los ajustes.

Consideraciones de seguridad para aplicaciones en tiempo real

Wisej.NET evita muchos de los problemas tradicionales de concatenar cadenas HTML, porque sincroniza controles del servidor con widgets del navegador en lugar de construir páginas con cadenas HTML arbitrarias. Aun así, los desarrolladores pueden introducir riesgo de XSS si muestran como HTML contenido proporcionado por el usuario. Los controles y mensajes que admiten HTML deben tratarse con cuidado. Mantén codificados los títulos de tickets, comentarios, nombres y mensajes introducidos por los usuarios, salvo que exista un proceso de saneamiento claro.

Detrás de un proxy inverso o de un balanceador de carga, el tratamiento de la IP del cliente también debe ser deliberado. Solo se debe confiar en las cabeceras reenviadas (forwarded headers) cuando la aplicación está detrás de un proxy de confianza que sanea las cabeceras entrantes. De lo contrario, un atacante puede falsificar sus valores.

Las aplicaciones en tiempo real también deben proteger los recursos del servidor. Establece límites para las sesiones, las colas y la frecuencia de eventos. No permitas que un usuario cree trabajos en segundo

plano o suscripciones sin límite.

Observabilidad

Una aplicación en tiempo real necesita logs y métricas que expliquen qué ocurrió después de que terminara la petición original del usuario. Incluye campos de correlación:

- id del trabajo;
- id de usuario o marcador de sesión anónima;
- client id o id de sesión cuando proceda;
- nombre de la operación;
- tipo de evento;
- instancia de servidor;
- duración y resultado;
- número de actualizaciones de interfaz enviadas o aplicadas.

En TicketOps Live, cada trabajo de importación debe tener un id de trabajo y cada evento del hub debe tener un id de evento. La interfaz puede mostrar un estado simplificado, pero los logs deben permitir al personal de soporte reconstruir la secuencia.

Lista de comprobación de rendimiento

Antes del despliegue, revisa estas preguntas:

1. ¿Tiene cada bucle en segundo plano una condición de parada?
2. ¿Cancela su suscripción cada suscripción a eventos globales?
3. ¿Se agrupan o limitan las actualizaciones de la interfaz?
4. ¿Se agrupan los eventos del modelo de alta frecuencia?
5. ¿Se actualizan las colecciones enlazadas en el contexto de la sesión?
6. ¿Se usan los campos estáticos solo para el estado global?
7. ¿Son por sesión los contadores, filtros y selecciones específicos de la sesión?
8. ¿Se capturan y registran las excepciones dentro de las tareas en segundo plano?
9. ¿Se comporta la aplicación de forma aceptable cuando WebSocket no está disponible?
10. ¿Mantiene el balanceador de carga cada sesión en la misma instancia de servidor?

Visión general del proyecto final

El proyecto final es una revisión de producción y la finalización de TicketOps Live. Los estudiantes deben demostrar:

- actualizaciones de estado en directo;
- progreso de la importación en segundo plano con cancelación y gestión de errores;
- actualizaciones en directo de la cuadrícula de tickets;
- notificaciones de tickets multiusuario;
- estrategia de polling como alternativa;

- limpieza de las sesiones;
- lista de comprobación de despliegue y observabilidad.

El proyecto final no se evalúa por el número de controles en pantalla, sino por la arquitectura: propiedad, contexto, ciclo de vida, cadencia de actualización y preparación para producción.



Punto de control

Al terminar este módulo, el estudiante debe poder explicar cómo una aplicación Wisej.NET en tiempo real pasa de ser una demo a una aplicación desplegable: ruta WebSocket verificada, enrutamiento de sesiones estable, alternativa definida, suscripciones limpiadas, actualizaciones limitadas, datos protegidos y operaciones observables.

Refuerzo para producción, despliegue y proyecto final

Qué construirás en el laboratorio práctico, las tareas, la implementación sugerida y los criterios de aceptación.



Laboratorio práctico

Completa TicketOps Live y prepáralo para una revisión de arquitectura de producción.

Funcionalidades requeridas

1. Franja de estado en directo del M01.
2. Diagnóstico de la sesión del M02.
3. Monitor de importación en segundo plano del M03.
4. Controles de cadencia y polling del M04.
5. Tablero de tickets en directo del M05.
6. **TicketHub** multiusuario del M06.
7. Lista de comprobación de producción y revisión de la configuración del M07.

Entregable de arquitectura

Redacta una nota de arquitectura de una página que indique:

- ✓ qué estado es específico de la sesión;
- ✓ qué estado es global;
- ✓ dónde empieza el trabajo en segundo plano;
- ✓ cómo llegan las actualizaciones de interfaz al navegador;
- ✓ cómo funciona la cancelación;
- ✓ cómo se eliminan las suscripciones;
- ✓ qué ocurre cuando WebSocket no está disponible;
- ✓ qué debe admitir el entorno de despliegue.

Guion de la demo

1. Abre TicketOps Live en dos sesiones del navegador.
2. Inicia el heartbeat.
3. Inicia una importación y muestra el progreso.
4. Cancela una importación.
5. Inicia otra importación y fuerza un fallo simulado.
6. Genera eventos de ticket nuevos y muestra que ambas sesiones se actualizan.
7. Aplica un filtro en una sesión y muestra que la otra sesión es independiente.
8. Desactiva el modo en directo y explica el comportamiento del polling y de la cadencia.

9. Presenta la lista de comprobación de producción.

Criterios de aceptación

- ✓ La aplicación se ejecuta sin bucles ni suscripciones duplicados.
- ✓ La interfaz sigue respondiendo durante el trabajo en segundo plano.
- ✓ Cada operación de larga duración tiene estados de finalización, cancelación y fallo.
- ✓ El hub multiusuario no guarda referencias directas a páginas ni a controles.
- ✓ Las cuadrículas enlazadas se actualizan de forma incremental.
- ✓ El estudiante sabe defender todas las suposiciones de despliegue.